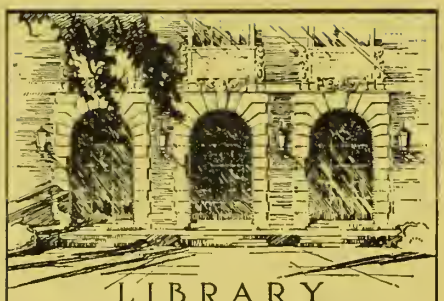


NUCLEOL - A MINIMAL LIST PROCESSOR

by

J. Nievergelt, F. Fischer, M. I. Irland and J. R. Sidlo

April, 1969

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. 324

NUCLEOL - A MINIMAL LIST PROCESSOR

by

J. Nievergelt, F. Fischer, M. I. Irland and J. R. Sidlo

April 1969

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

Paper presented at the Purdue Centennial Year Symposium on Information Processing, April 28, 1969.

This work was supported by the Department of Computer Science at the University of Illinois, the National Science Foundation under NSF Grant GJ 217, and by BUILD, which sponsors cooperation between the Universities of Illinois and Colorado.

NUCLEOL - A Minimal List Processor

J. Nievergelt, F. Fischer, M. I. Irland, J. R. Sidlo*
Department of Computer Science, University of Illinois

Abstract

NUCLEOL is a low-level list processor designed as a basis in terms of which higher-level list- and string-processing languages could be implemented easily and efficiently. Hence its design aims at:

a) Simplicity
b) Complete and concise description
c) General data structures and a small, well-chosen set of primitive operations
d) A scheme for implementation which makes it easy to transfer the system from one computer to another.

The system is currently implemented as a PL/1 program.

1. Background and purpose

The project to be described has its origin in our experiences with transferring a symbol manipulation language from one computer to others. The language in question is EOL [1,2,3], which was first implemented on the Polish computer ZAM at the Institute for Mathematical Machines in Warsaw and later on an IBM 7094 and an IBM 360 at the University of Illinois. The specific features of EOL are not important for the purpose of this paper. They have, however, strongly influenced our design of NUCLEOL.

Implementations of different list processing languages have many common features--indeed, their very name refers to a particular scheme for organizing storage. In most implementations, however, these common features are usually, but unnecessarily, tied to a particular language and it appears not to be common practice that different list-processing languages share the same basic subroutines.

Our guideline in designing NUCLEOL was to draw an explicit dividing line between what we consider to be "high-level features" in which list-processing languages tend to differ a great deal, and the "low-level features" common to most of them: and to have NUCLEOL provide most of the latter and so, in a sense, simulate a computer specifically adapted to list processing.

An important qualification must be made at this point. The name "list" covers several data structures among which it is useful to distinguish, in order of increasing generality:

* M. I. Irland is currently with the University of Waterloo and J. R. Sidlo with Computer Sciences Corporation.

strings or linear lists (one-way or two-way)

trees (lists without shared sublists)

graphs (structures whose elements are linked to each other in arbitrary ways)

Probably the only thing common in handling all of these data structures is the ability to define fields and manipulate pointers. There are languages like L^6 [4] that are aimed at this level. We think that drawing the dividing line at the level of fields and pointers leaves a much greater part of the implementation of a high-level list-processing language above than below the line, and we aimed at easing the burden of implementing list-processing languages to a greater extent. In particular, we wanted NUCLEOL itself to take care of the organization of lists in terms of data fields and pointers, so that a user would not have to refer explicitly to pointers to carry out such list-oriented operations as insertion and deletion of sublists (but could, e.g., say something like "insert list x at point y in list z").

Aiming at this level, however, forced us to renounce the generality of graphs as data structures. It is in the restricted case of trees that we felt there were sufficiently general and efficient common operations that would warrant our effort.

Our goal in designing NUCLEOL can now be stated as follows: To provide a system, as simple as possible, which is sufficient, in a practical sense, so that any list-processing language which operates on tree-structured data can be implemented in terms of it easily and efficiently. It became crucially important then, that the description of NUCLEOL itself be complete, so there would be no misunderstandings to a person who studied it sufficiently deeply. And that the implementation of NUCLEOL itself be simple.

We will discuss at the end of this paper to what degree we consider having achieved this goal.

2. Informal Description

NUCLEOL programs as well as data are well-formed strings (abbreviated as wfs) of units called constituents. A constituent carries the following information: its type, an attribute, and (with the exception of parenthesis constituents mentioned below) data. Among the various types of constituents there are two, the left parenthesis ¢(and the right parenthesis ¢) which occur in a wfs in a balanced way. Hence it is convenient to introduce a unit called a block, which is either a single constituent (other than a parenthesis) or an appropriate string enclosed in parentheses. Because of this block structure a wfs can be interpreted as being organized as a tree as well as a linear string.

Each wfs is accessed by means of a unique constituent ¢S called the scanner, which can be moved around in a manner convenient for both of the interpretations of a wfs as a linear list or a tree. The scanner gives access to the two blocks to its left and right (if present), and also designates the two gaps to its left and right (where a new block may be inserted). The scanner carries a name as its data, which is also the name of the wfs accessed by this scanner.

Apart from parentheses and scanner, there is one more type of constituent which relates to the structure of wfs's called a reference constituent ¢R. It may occur anywhere in a wfs and refer to any wfs, either in its entirety or to the blocks or gaps near the scanner.

The remaining types are data constituents, of which there are three, namely ¢B (bitstrings), ¢C (characterstrings) and ¢D (numbers), and finally a parameter constituent ¢P, which is used to represent formal arguments in macros.

A NUCLEOL state is a set of wfs's no two of which have the same name, and exactly one of which has a scanner whose attribute characterizes it as the execution scanner. The syntax of NUCLEOL is given by a set of rules (mostly in Backus Naur Form, but the sentence above is also part of the definition) which defines what a NUCLEOL state is.

The semantics is defined by a function NUCSTEP, which assigns to some of the states a next state. A NUCLEOL execution is a sequence of states each one of which gets transformed by NUCSTEP into its successor. If there is a last state, then either NUCSTEP is undefined on it, or during the past transition one of a small number of stop conditions must have occurred.

The following example shows a NUCLEOL state consisting of three wfs's (represented in what we call the reference language):

```
¢(X       ¢S 'SINK'       ¢)X

¢(X ¢B '00111' ¢(  ¢D  '-17'  ¢C 'STRING OF ARBITRARY LENGTH' ¢) ¢S 'SOURCE' ¢)X
                 \_____________________________________/
                          block at left of scanner ¢S 'SOURCE'

¢SN 'PROGRAM' ¢(XN  ¢CK 'MOVE'  ¢RL 'SOURCE'  ¢RL 'SINK' ¢)XW
```

The wfs SINK is as small as it can be, since every wfs must contain at least a pair of external parentheses (distinguished by an attribute X) and a scanner. The scanner ¢S 'SOURCE' has a block to its left but none to its right.

The scanner named PROGRAM is in its external position (i.e., outside the external parentheses. This position is distinguished by the fact that (by definition) the block to the right of the scanner is the same as the block to its left, namely, the entire wfs exclusive of the scanner. I.e., wfs are considered to be circular. This scanner is also distinguished by its attribute N (for neutral) to be the execution scanner, and its motion represents the flow of control in the program.

The ¢C constituent inside wfs PROGRAM has an attribute K (for keyword), which marks the beginning of the instruction ¢CK 'MOVE' ¢RL 'SOURCE' ¢RL 'SINK'. Each of the two ¢R constituents has an attribute L (for left), and execution of this instruction causes the block currently to the left of scanner ¢S 'SOURCE' to be deleted and inserted in the gap to the left of scanner ¢S 'SINK'.

Let us now trace the sequence of successive states generated by repeated application of the function NUCSTEP on the state described above.

First step: the attribute (N) of the execution scanner is matched against the protection attribute (N) of the ¢(constituent to its right. Because

they match, the scanner enters the block, i.e., is shifted past the constituent to its right (if the attributes do not match, the execution scanner skips around the block to its right).

Second step: the MOVE instruction is executed, which deletes the block ¢(¢D '-17' ¢C 'STRING OF ARBITRARY LENGTH' ¢) from wfs SOURCE and inserts it to the left of scanner ¢S 'SINK'. Thereafter, the execution scanner is placed to the right of the instruction just executed.

Third step: the execution scanner, which still has an attribute N, cannot pass through the ¢) constituent with attribute W. This causes the scanner to bounce to the corresponding left parenthesis, where it finds itself again in front of the MOVE instruction.

Fourth step: the MOVE instruction is executed again, and this time the single constituent ¢B '00111' is deleted from wfs SOURCE and inserted in the gap to the left of scanner ¢S 'SINK'. Then the execution scanner moves past the instruction.

Fifth step: execution scanner bounces again.

Sixth step: the execution scanner attempts to execute the MOVE instruction a third time. During evaluation of its first argument ¢RL 'SOURCE', however, it is found out that now there is no block to the left of the scanner ¢S 'SOURCE'. This causes the execution scanner to skip the MOVE instruction as before, but now its attribute changes from N to W (our mnemonic for "something went wrong").

Seventh step: now that the attribute of the execution scanner matches the attribute of the parenthesis, the scanner passes through the ¢)XW, instead of bouncing. Since the execution scanner's exiting through an external parenthesis is a condition for stopping, the sequence terminates here.

Hence, the one-instruction program above has the effect of deleting all the blocks (at a given level of nesting -- in this case all first-level blocks) from wfs SOURCE and inserting them in reversed order in wfs SINK.

NUCLEOL has 15 instructions (or 16, depending on how they are counted), of which we now give a rather abbreviated description (compare this with the syntactic description in the next section).

Structural Operations

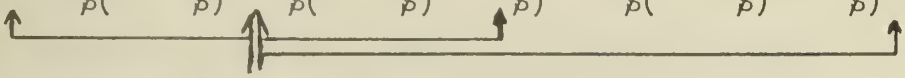
MOVE In addition to the motion of blocks described earlier, this instruction also serves the purpose of returning blocks and entire wfs's to the free storage list (e.g., ¢CK 'MOVE' ¢R 'WFS' ¢R 'SYS-FREE'), and of input and output (e.g., ¢CK 'MOVE' ¢R 'SYSINPUT' ¢R " adds a named wfs to the current NUCLEOL state from the system's input device).

COPY Acts in the same way as MOVE, except that the original is copied, not deleted.

SHFT Shifts a scanner one constitutent to the left or right, and hence may be used to enter and leave blocks (the MOVE instruction is used to skip blocks). Since wfs are considered to be circular, shifting a scanner from its external position in either direction makes sense.

RSTR Restores a scanner to one of three positions: its external position, to the right of the next outer left parenthesis, to the left of the next outer right parenthesis.

E.g.: ¢(¢(¢(¢) ¢(¢) ¢) ¢(¢) ¢)



Bitstring Operations

AND, OR, NOT generate a ¢B constituent whose bitstring is obtained by performing bitwise logical operation on the bitstrings in its arguments.

Characterstring Operations

CONC generates a ¢C constituent whose characterstring is the concatenation of the strings in its arguments.

SPLT splits the last character from a ¢C constituent and generates a new ¢C constituent from it.

Numerical Operations

ADD, SUB, MLT, DIV generate a ¢D constituent whose number is the result of performing an arithmetic operation on the numbers in the arguments.

Data Conversion

CVRT converts (if possible) a constituent of one type to a constituent of another type and/or attribute (e.g., ¢D to ¢C or ¢B and vice versa, ¢C to ¢R, etc.).

TEST

All of the above instructions set the attribute of the execution scanner to W if they cannot be performed. The last instruction, TEST, can set this attribute to one of four values, namely:

S test was successful

F failure (the comparison demanded by the test was carried out and the result was negative)

U undefined (the data to be compared was not of the proper type)

W wrong (the data to be compared could not be accessed)

Having more than two possible outcomes for a test is natural and very useful when accessing a data item is as much part of the test as comparing it once it has been found. The outcome indicates how far execution of the test could be carried out.

Jumps

Notice there seem to be no go-to-statements in this list of instructions. This is not quite true, as the instruction RSTR, when it refers to the execution scanner itself is a jump (of limited usefulness). Much more control is available by using the "bouncing and skipping" logic which depends on the protection attributes of parentheses and the attribute of the execution scanner.

There is, however, a hidden 16th instruction, which takes effect when the execution scanner finds itself just in front of a $R constituent, as in

$SN 'PROGRAM' $RR 'NEW'

The NUCSTEP function causes the following changes to occur in the state.

a) Shift the scanner PROGRAM past the reference constituent $RR 'NEW'

b) Reset the attribute of scanner PROGRAM (to "blank") so it is no longer the execution scanner.

c) Set the attribute of scanner NEW to N, so it becomes the execution scanner.

Notice that execution continues wherever the scanner NEW happened to be. By executing the reference $R 'NEW' instead of $RR 'NEW', the scanner NEW would have been reset to its external position before exchanging control.

It is clear that with this facility, and given that reference constituents can be operated upon, such devices as subroutine call and return, coroutine jumps, and switches are easily programmable.

We don't present this as evidence that labels and go-to statements are obsolete (maybe Dijkstra would? - see [5]). We considered seriously having label constituents and allowing references to them. In NUCLEOL, however, such labels would necessarily be dynamic, and the overhead associated with their use (e.g., what happens when you copy a label?) did not seem consistent with our aims of simplicity. Not having labels in NUCLEOL, of course, does not imply that there could not be labels in a language based on it.

3. Formal Definition

Describing a programming language and defining it are two very different things. In a description to someone unfamiliar with a language one wants to stress a few highlights and avoid burdening his memory with details. This is what we have attempted to do in the previous section. In a definition, on the other hand, one has to say everything there is to say. Because of the intended use of NUCLEOL as a basis in terms of which other languages may be implemented, we felt it necessary to attempt at least to provide a complete rigorous definition of the language.

Below is a complete definition of the syntax of NUCLEOL, mostly in (a slightly modified) Backus-Naur Form but also containing some English sentences (for convenience and, in one case, necessity). The notation <something * > means "one or more occurrences of <something>" and <something*?> means "zero or more occurrences of <something>".

NUCLEOL Syntax

<STATE> ::= A SET OF <WFS>'s NO TWO OF WHICH HAVE <$S>'s WITH THE SAME <WFS NAME> AND EXACTLY ONE OF WHICH HAS A <$S> WITH <SA> EQUAL TO N, S, F, U, OR W.

<WFS> ::= <$S> $(X <PA> <BLOCK*?> $)X <PA> |
$(X <PA> <BLOCK*?> <SBLOCK> <BLOCK*?> $)X <PA>

<SBLOCK> ::= <$S> | <$(> <BLOCK*?> <SBLOCK> <BLOCK*?> <$)>

```
<BLOCK> ::= <ORDINARY CONSTITUENT> | <$(> <BLOCK*?> <$)>
<CONSTITUENT> ::= <ORDINARY CONSTITUENT> | <$(> | <$)> | <$S>
<ORDINARY CONSTITUENT>  ::= <$B> | <$C> | <$D> | <$P> | <$R>
<$B> ::= $B<BA>'<BITSTRING>'
<$C> ::= $C<CA>'<CHARACTERSTRING>'
<$D> ::= $D<DA>'<NUMBER>'
<$P> ::= $P<TA>'<CHARACTERSTRING>'
<$R> ::= $R<RA>'<WFS NAME>'

<$(> ::= $(<PA>
<$)> ::= $)<PA>
<$S> ::= $S<SA>'<WFS NAME>'

<BA> ::= <BLANK> | S
<CA> ::= <BLANK> | K | M | S
<DA> ::= <BLANK> | S
<TA> ::= <BLANK> | B | C | D | R
<RA> ::= <BLANK> | L | R
<PA> ::= <BLANK> | N | S | F | U | W  OR COMBINATIONS OF N, S, F, U, W
<SA> ::= <BLANK> | N | S | F | U | W

<BITSTRING> ::= <BIT*?>
<BIT> ::= 0 | 1
<CHARACTERSTRING> ::= <CHARACTER*?>
<CHARACTER> ::= <BLANK> | <REST> | <LETTER> | <DIGIT>
<BLANK> ::= A SINGLE SPACE
<REST> ::= . | < | ( | + | & | ! | $ | * | ) | ; | ¬ | -
         | / | , | % | _ | > | ? | : | ≠ | ∂ | ' | = | "
<LETTER> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P
          | Q | R | S | T | U | V | W | X | Y | Z
<DIGIT> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<NUMBER> ::= <DIGIT*> | +<DIGIT*> | -<DIGIT*>
<WFS NAME> ::= SEQUENCE OF LETTERS, DIGITS, AND THE CHARACTER '_',
            BEGINNING WITH A LETTER AND NOT LONGER THAN 8 CHARACTERS.

<INSTRUCTION> ::= <MOVE> | <COPY> | <SHFT> | <RSTR> |
                  <CVRT> | <ADD> | <SUB> | <MLT> | <DIV> | <AND> | <OR> |
                  <NOT> | <CONC> | <SPLT> | <TEST>

<MOVE> ::= $CK'MOVE' <$R> <$R>
<COPY> ::= $CK'COPY' ( <BLOCK> <$R> | <$R> <$R> )
<SHFT> ::= $CK/SHFT' <SR>
<RSTR> ::= $CK'RSTR' ( <SR> | <WR> )
<CVRT> ::= $CK'CVRT' ( <T/A> | <SR> ) <SR>
<ADD>  ::= $CK'ADD' ( <$D> | <SR> ) ( <$D> | <SR> ) <SR>
<SUB>  ::= $CK'SUB' ( <$D> | <SR> ) ( <$D> | <SR> ) <SR>
<MLT>  ::= $CK'MLT' ( <$D> | <SR> ) ( <$D> | <SR> ) <SR>
<DIV>  ::= $CK'DIV' ( <$D> | <SR> ) ( <$D> | <SR>  <SR>
<AND>  ::= $CK'AND' ( <$B> | <SR> ) ( <$B> | <SR> ) <SR>
<OR>   ::= $CK'OR'  ( <$B> | <SR> ) ( <$B> | <SR> ) <SR>
<NOT>  ::= $CK'NOT' ( <$B> | <SR> ) <SR>
<CONC> ::= $CK'CONC' ( <$C> | <SR> ) ( <$C> | <SR> ) <SR>
<SPLT> ::= $CK'SPLT' <SR> <SR>
<TEST> ::= $CK'TEST' ( <$B> | <$C> | <$D> | <$P> | <SR> ) ( <TEST MODE> |
          <SR> ) ( <$B> | <$C> | <$D> | <$P> | <SR> )

<SR>   ::= $RL'<WFS NAME>' | $RR'<WFS NAME>'
<WR>   ::= $R '<WFS NAME>'
```

```
<T/A> ::= ¢C'B/<BA>' | ¢C'C/<CA>' | ¢C'D/<DA>' | ¢C'R/<RA>' | ¢C'P/<TA>' |
          ¢C' /<RA>' | ¢C' /<PA>' |
          ¢C' /<BA>' | ¢C' /<CA>' | ¢C' /<DA>' | ¢C' /<TA>'

<TEST MODE> ::= ¢C'  = '   |  ¢C' ¬= '   |  ¢C' <  '   |
                ¢C' <= '   |  ¢C' >  '   |  ¢C' >= '   |
                ¢C'D =A'   |  ¢C'D¬=A'   |  ¢C'A =D'   |  ¢C'A¬=D'  |
                ¢C'A =A'   |  ¢C'A¬=A'   |  ¢C'T =D'   |  ¢C'T¬=D'  |
                ¢C'A =T'   |  ¢C'A¬=T'   |  ¢C'T =T'   |  ¢C'T¬=T'  |
                ¢C'T =T'   |  ¢C'T¬=T'   |  ¢C'D =D'   |  ¢C'D¬=D'  |
                ¢C'T =A'   |  ¢C'T¬=A'
```

While there are well-established tools for the definition of the syntax of programming languages, the situation is completely different with respect to semantics.

We insisted that the definition should serve the dual purpose of defining NUCLEOL to humans and to machines. This principle is not often taken into consideration. It is correct that any compiler or interpreter defines a language to a particular computer completely, but this is not of much use to somebody who must implement the language on a new machine.

A review of earlier attempts to define programming languages indicated to us that McCarthy's approach ([6, 7] and other papers), would be best suited to serve our dual purpose. Hence a definition of NUCLEOL was written which is, at the same time, a PL/1 program for an interpreter. PL/1 was chosen because, among well-known high-level languages, it offers the greatest flexibility of notation, which is an important point if a program is to be its own documentation.

The interpreter which resulted currently consists of about 1500 PL/1 statements. We estimate that through "tight coding" this number could be reduced to 1000, but our aim was clear documentation and avoidance of all "tricky" programming.

Only the top part of the interpreter, which consists of about 400 statements, is part of the formal definition of NUCLEOL. It is written in terms of about 50 basic predicates and functions, listed below. The remaining 1000 statements implement these predicates and functions and they are too detailed and machine-dependent (in this case, PL/1-dependent) to be very enlightening.

NUCLEOL Basic Functions and Predicates

BASIC PREDICATES:

```
IS_STATE(STATE) ;
IS_WFS(WELL_FORMED_STRING) ;
IS_BLOCK(BLOCK) ;
IS-CONSTITUENT(CONSTITUENT) ;
IS_TYPE(TYPE) ;
IS_DIRECTION(DIRECTION) ;
IS_BITS(BITSTRING) ;
IS-CHRS(CHARACTERSTRING) ;
IS-NUMB(NUMBER) ;
IS_NAME(NAME) ;
IS_CONVERTIBLE(CONVERSION_MODE,CONSTITUENT) ;
CAN_PASS(SCANNER_ATTRIBUTE,PARENTHESIS-ATTRIBUTE) ;
TESTS(TEST_MODE,CONSTITUENT1,CONSTITUENT2)  ;
```

```
   STATE LEVEL FUNCTIONS :

EXEC(STATE)=WFS_NAME ;
WFS-NAMED(WFS_NAME)=WFS ;
KILL(WFS-NAME,STATE)=NEW_STATE ;
CREATE(WFS_NAME,WFS,STATE)=NEW-STATE ;

   WFS LEVEL FUNCTIONS :

BLOCK_AT(DIRECTION,WFS)=BLOCK ;
CONSTITUENT-AT(DIRECTION,WFS)=CONSTITUENT ;
DELETE(DIRECTION,WFS)=NEW_WFS ;
INSERT(DIRECTION,BLOCK,WFS)=NEW_WFS
SKIP-BLOCK(DIRECTION,WFS)=NEW_WFS ;
SHIFT(DIRECTION,WFS)=NEW_WFS ;
RESTORE(WFS)=NEW_WFS ;

   CONSTITUENT LEVEL FUNCTIONS :

TYPE_OF(CONSTITUENT)=TYPE ;
ATTR_OF(CONSTITUENT)=ATTRIBUTE ;
BITS_IN(CONSTITUENT)=BITSTRING ;
CHRS_IN(CONSTITUENT)=CHARACTERSTRING ;
NUMB_IN(CONSTITUENT)=NUMBER ;
NAME-IN(CONSTITUENT)=WFS-NAME ;
CONVERT_DATA(CONVERSION_MODE,CONSTITUENT)=NEW-CONSTITUENT ;
SET_ATTR(ATTRIBUTE,CONSTITUENT)=NEW-CONSTITUENT ;

ADDS(CONSTITUENT1,CONSTITUENT2)=NEW_CONSTITUENT ;
SUBS(CONSTITUENT1,CONSTITUENT2)=NEW_CONSTITUENT ;
MLTS(CONSTITUENT1,CONSTITUENT2)=NEW_CONSTITUENT ;
DIVS(CONSTITUENT1,CONSTITUENT2)=NEW-CONSTITUENT ;
ANDS(CONSTITUENT1,CONSTITUENT2)=NEW_CONSTITUENT ;
 ORS(CONSTITUENT1,CONSTITUENT2)=NEW_CONSTITUENT ;
NOTS(CONSTITUENT1)=NEW-CONSTITUENT ;
CONCS-CHRS(CONSTITUENT1,CONSTITUENT2)=NEW_CONSTITUENT ;
SPLIT_CHRS1(CONSTITUENT)=NEW_CONSTITUENT ;
SPLIT_CHRS2(CONSTITUENT)=NEW_CONSTITUENT ;

L_NEBR(CONSTITUENT)=OTHER_CONSTITUENT ;
R_NEBR(CONSTITUENT)=OTHER_CONSTITUENT ;
MATCH_PAREN(PARENTHESIS)=OTHER_PARENTHESIS ;

   SUBCONSTITUENT LEVEL FUNCTIONS :

OPPOSITE(DIRECTION)=NEW-DIRECTION ;
```

As an example of the definitional part of the interpreter, we show below the top level of the function NUCSTEP discussed earlier, and the interpreter's "main loop" which calls NUCSTEP.

```
   DO WHILE (IS_STATE(STATE)) ;
      STATE = NUCSTEP(STATE) ;
   END ;
```

```
   NUCSTEP: PROCEDURE(STATE) ;
NXT = CONSTITUENT_AT_RIGHT(EXEC_SCANNER);
IF TYPE_OF(NXT) = ¢C & ATTR_OF(NXT) = ∂K THEN RETURN(EXECUTE-INSTR(STATE));
IF TYPE_OF(NXT) = ¢LEFT_PAREN | TYPE_OF(NXT) = ¢RIGHT_PAREN THEN DO ;
   IF CAN_PASS(ATTR_OF(EXEC_SCANNER),ATTR_OF(NXT)) THEN DO;
      ATTR_OF(EXEC_SCANNER) = ∂N;
      RETURN(SHIFT(RIGHT,EXEC_SCANNER));
   END;
   CONSTITUENT_AT_RIGHT(EXEC_SCANNER) = R_NEBR(MATCH_PAREN(NXT));
   RETURN(STATE);
END;
IF TYPE_OF(NXT) = ¢R THEN DO;
   IF NAME_IN(NXT) = 'SYS_STOP' THEN GO TO STOP;
   STATE = SHIFT(RIGHT,EXEC_SCANNER);
   ATTR_OF(EXEC_SCANNER) = ∂;
   WFS = WFS_NAMED(NAME_IN(NXT));
   ATTR_OF(WFS) = ∂N;
   EXEC_SCANNER = WFS; /* CHANGE EXECUTION SCANNER */
   IF ATTR_OF(NXT) = ∂ THEN RETURN(RESTORE(WFS));
   RETURN(STATE);
END;
RETURN(SHIFT(RIGHT,EXEC_SCANNER)); /* IN ALL OTHER CASES */
END NUCSTEP;
```

The definition of NUCLEOL is completed by a set of about 60 postulates which relate the basic predicates and functions to each other. Here is a sample.

NUCLEOL Postulates

```
IS_TYPE(TYPE) <=> TYPE=¢B
                | TYPE=¢C
                | TYPE=¢D
                | TYPE=¢P
                | TYPE=¢R
                | TYPE=¢LEFT_PAREN
                | TYPE=¢RIGHT_PAREN  ;

IS_DIRECTION(D) <=> D=LEFT | D=RIGHT ;
OPPOSITE(LEFT)=RIGHT ;
OPPOSITE(RIGHT)=LEFT ;
IS_DIRECTION(OPPOSITE(D)) <=> IS_DIRECTION(D) ;

IS_TYPE(TYPE_OF(C)) <=> IS_CONSTITUENT(C) ;
IS_BITS(BITS_IN(C)) <=> IS_CONSTITUENT(C) ;
IS_CHRS(CHRS_IN(C)) <=> IS_CONSTITUENT(C) ;
IS_NUMB(NUMB_IN(C)) <=> IS_CONSTITUENT(C) ;
IS_NAME(NAME_IN(C)) <=> IS_CONSTITUENT(C) ;

IS_CONSTITUENT(ADDS(C1,C2)) <=> TYPE_OF(C1)=¢D&TYPE_OF(C2)=¢D ;
IS_CONSTITUENT(ADDS(C1,C2)) => TYPE_OF(ADDS(C1,C2))=¢D ;

IS_CONVERTIBLE(CONVERSION_MODE,CONSTITUENT)
     => IS_CONSTITUENT(CONVERT_DATA(CONVERSION_MODE,CONSTITUENT)) ;

IS_CONSTITUENT(CONSTITUENT) => IS_BLOCK(CONSTITUENT)
                  | TYPE_OF(CONSTITUENT)=¢LEFT_PAREN
                  | TYPE_OF(CONSTITUENT)=¢RIGHT_PAREN  ;
```

```
IS_CONSTITUENT(CONSTITUENT_AT(DIRECTION,WFS))
    <=> IS_DIRECTION(DIRECTION) & IS_WFS(WFS) ;

RESTORE(RESTORE(WFS))=RESTORE(WFS) ;
IS_DIRECTION(D) & IS_WFS(WFS)
     => RESTORE(WFS)=RESTORE(SHIFT(D,WFS)) ;

IS_BLOCK(BLOCK_AT(DIRECTION WFS))
     => INSERT(DIRECTION,BLOCK_AT(DIRECTION,WFS), DELETE(DIRECTION,WFS))=WFS ;
```

To summarize: Our definition of NUCLEOL consists of about 400 PL/1 statements which are part of an interpreter, and about 60 postulates which relate the functions to each other in terms of which the interpretive part of the definition is written. Needless to say, we would have liked to prove some sort of completeness of this definition, but we just didn't know how to go about doing this.

4. Conclusion

We consider having been successful in reducing a programming language of potentially great complexity (because of the data structures involved) to a small yet practically usable core, whose parts fit into a conceptual system with few basic notions. The adequacy of the instruction set was tested during the design stage by writing a macrogenerator for NUCLEOL, in NUCLEOL.

We have not yet reached a definite opinion concerning the practical feasibility of a formal definition of programming languages, even one as simple as NUCLEOL. McCarthy's approach (which, incidentally, is the main base for an attempt at the formal definition of PL/1 by a group at the IBM laboratory in Vienna (see [8], and many reports)) appeared to amount essentially to "good programming"--e.g., identify the basic functions in terms of which the interpreter should be written, distinguish carefully among different levels of activity (in our case: operations on the state, on a wfs, a block, a constituent, and finally on the data contained in a constituent).

5. Current Work

One of the guiding lights in the design of NUCLEOL was its applicability to tree transformations as they occur in linguistic analysis, particularly in testing transformational grammars. Such a system, in which tree transformations can be specified by patterns and replacements, and in which subtrees, which match the pattern, are replaced recursively is currently being written in NUCLEOL.

Lastly, for NUCLEOL to serve its purpose it is important that it may be implemented easily on other machines, and that it may run efficiently. Having an interpreter written in PL/1 solves the first problem for installations which have a PL/1 compiler, but hardly the second one.

Our aim in writing the PL/1 interpreter was mainly one of documentation. It is intended that efficient implementations of NUCLEOL will be obtained by using this interpreter not as a PL/1 program, but as an input to a macro processor. For each of the PL/1 constructs used (care was exercised to limit this set as much as possible) a macro has to be defined in the target language. This leaves an implementor free to choose the internal representation of the NUCLEOL data structure to be the most efficient on his particular computer.

W. M. Waite of the University of Colorado is currently working on the implementation of NUCLEOL on a CDC 6400 and a (decimal) Librascope computer using this scheme and his Mobile Programming System [9].

References

1. Lukaszewicz, L. "EOL - A Symbol Manipulation Language," The Computer Journal, Vol. 10, No. 1, May, 1967.

2, 3. Lukaszewicz, L. and Nievergelt, J. "EOL-Report" and "EOL Programming Examples," U. of Illinois, DCS Reports Nos. 241, 242, Sept., 1967.

4. Knowlton, K. C. "A Programmer's Description of L^6," C. ACM, Vol. 9, No. 8, Aug., 1966.

5. Dijkstra, E. "Go To Statement Considered Harmful," C. ACM, Vol. 11, No. 3, March, 1968 (letter to the editor).

6. McCarthy, J. "A Formal Description of a Subset of Algol," in "Formal Language Description Languages" (ed., Steel), pp. 1-12, North Holland, 1966.

7. McCarthy, J. and Painter J. "Correctness of a compiler for arithmetic expressions" in "Mathematical Aspects of Computer Science," Proc. Symp. Applied Math, Vol. 19, American Math. Society, 1967.

8. Bandat, K. "On the formal definition of PL/1," Proc. AFIPS SJCC, 1968, pp. 363-373.

9. Orgass, R. J. and Waite, W. M. "A Base for a Mobile Programming System," IBM Research Paper, RC- 1952, Nov., 1967.

10. Sidlo, J. R. "NUCLEOL - The Basis for the List-Processing Language EOL-4," M.S. Thesis, U. of Illinois, Aug., 1968.

11. Irwin-Zarecki, M. (I. Irland). "NUCLEOL as a Formal System," M.S. Thesis, U. of Illinois, Feb., 1969.

Acknowledgments

We are grateful to Professor L. Lukaszewicz of the Polish Academy of Sciences, W. M. Waite of the University of Colorado, and B. D. Weathers of the University of Missouri for important discussions on the design of NUCLEOL; and to Mr. Kiyoshi Maruyama for his help in programming and debugging.

Our work was supported by the Department of Computer Science at the University of Illinois, the National Science Foundation under NSF Grant GJ 217, and by BUILD, which sponsors cooperation between the Universities of Illinois and Colorado.